Lecture 2
Computer Systems: 
Program and its Execution

Zhiqiang Lin
Zhiqiang.lin@utdallas.edu
The University of Texas at Dallas
January 13rd, 2016
What will **executing** this program do?

```c
#include <stdio.h>

void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}

void main(int argc, char *argv[]){
    int x;
    x = 4300 + 93;
    answer(argv[1], x);
}
```

To answer the question

“Is this program safe/secure/vulnerable?”

We need to know

“What will executing this program do?”

Understanding the **compiler** and **machine semantics** are key.
Agenda

• Compilation Workflow

• **x86** Execution Model
  – Basic Execution
  – Memory Operation
  – Control Flow
  – Memory Organization
```c
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}
void main(int argc, char *argv[]){
    int x;
    x = 4300 + 93;
    answer(argv[1], x);
}
```

Compilation

Alice

0011010
1101010
1000101

The *compiler* and *machine* determines the semantics

Alice, the answer is 4393
“Compiled Code”

Source Language → Compilation → Target Language

Input

Output
“Interpreted Code”

Source Language Input

Interpretation

Output
Source Language
4393.c in C

Compilation

Target Language
4393 in x86

Pre-processor (cpp)

4393.c

Compiler (cc1)

Assembler (as)

Linker (ld)

4393.cinC
4393.c
#include <stdio.h>
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}
...

#include expansion
#define define substitution
Pre-processor (cpp) → Compiler (cc1) → Assembler (as) → Linker (ld)

```
#include <stdio.h>
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}
...
```

$ gcc -S

Creates Assembly
gcc –S 4393.c outputs 4393.s

_answer:
Leh_func_begin1:
    pushq %rbp
Ltmp0:
    movq %rsp, %rbp
Ltmp1:
    subq $16, %rsp
Ltmp2:
    movl %esi, %eax
    movq %rdi, -8(%rbp)
    movl %eax, -12(%rbp)
    movq -8(%rbp), %rax
...
Pre-processor (cpp) → Compiler (cc1) → Assembler (as) → Linker (ld)

_answer:
Leh_func_begin1:
    pushq %rbp
Lttmp0:
    movq %rsp, %rbp
Lttmp1:
    subq $16, %rsp
Lttmp2:
    movl %esi, %eax
    movq %rdi, -8(%rbp)
    movl %eax, -12(%rbp)
    movq -8(%rbp), %rax
....

$ as <options>

Creates object code

4393.s
$ ld <options>

Links with other files and libraries to produce an exe
Disassembling

• Today: using objdump (part of binutils)
  – objdump –D <exe>

• If you compile with “-g”, you will see more information
  – objdump –D –S
The program **binary** (aka executable)

Final executable consists of several **segments**

- Text for code written
- Read-only data for constants such as “hello world” and globals
  - ...

$ readelf –S <file>
Machine Instruction Example

- **C Code**
  - Add two signed integers

- **Assembly**
  - Add 2 4-byte integers
    - “Long” words in GCC parlance
    - Same instruction whether signed or unsigned
  - Operands:
    - \texttt{x}: Register \%eax
    - \texttt{y}: Memory \texttt{M[\%ebp+8]}
    - \texttt{t}: Register \%eax
      - Return function value in \%eax

- **Object Code**
  - 3-byte instruction
  - Stored at address \texttt{0x80483ca}
Agenda

• Compilation Workflow
• x86 Execution Model
  – Basic Execution
  – Memory Operation
  – Control Flow
  – Memory Organization
Basic Execution

File system

Binary
- Code
- Data
- ...

Fetch, decode, execute

Processor

Process Memory
- Stack
- Heap

read and write
x86 Processor

- EIP: Address of next instruction
- EFLAGS: Condition codes
- EAX
- ECX
- EDX
- ESP
- EBP
- ESI
- EDI

Purpose:

- General
Registers have up to 4 addressing modes

1. Lower 8 bits
2. Mid 8 bits
3. Lower 16 bits
4. Full register
x86 Implementation

- **EIP** is incremented after each instruction
- Instructions are different length
- EIP modified by CALL, RET, JMP, and cond. JMP
x86 Instruction Set

• Instructions classes:
  – Data Movement: MOV, PUSH, POP, ...
  – Arithmetic: TEST, SHL, ADD, ...
  – I/O: IN, OUT, ...
  – Control: JMP, JZ, JNZ, CALL, RET
  – String: REP, MOVSB, ...
  – System: IRET, INT, ...

• Volume 2A: Instruction Set Reference, A-M
  Volume 2B: Instruction Set Reference, N-Z
  – Intel syntax: OP DST, SRC
  – AT&T (gcc/gas) syntax: OP SRC, DST
## Basic Ops and AT&T vs Intel Syntax

### AT&T Syntax

<table>
<thead>
<tr>
<th>Meaning</th>
<th>AT&amp;T</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx = eax</td>
<td>movl %eax, %ebx</td>
<td>mov ebx, eax</td>
</tr>
<tr>
<td>eax = eax + ebx</td>
<td>addl %ebx, %eax</td>
<td>add eax, ebx</td>
</tr>
<tr>
<td>ecx = ecx &lt;&lt; 2</td>
<td>shr $2, %ecx</td>
<td>shr ecx, 2</td>
</tr>
</tbody>
</table>

- AT&T is at odds with assignment order. It is the default for objdump, and traditionally used for UNIX.

- Intel order mirrors assignment. Windows traditionally uses Intel, as is available via the objdump ‘-M intel’ command line option.
Agenda

• Compilation Workflow
• x86 Execution Model
  – Basic Execution
  – Memory Operation
  – Control Flow
  – Memory Organization
x86: **Byte Addressable**

*It's convention:* lower address at the bottom

- Address 0 holds 1 byte
- Address 1 holds 1 byte
- Address 2 holds 1 byte
- Address 3 holds 1 byte

I can fetch bytes at any address

Memory is just like using an array!

**Alternative:** **Word addressable**

**Example:** For 32-bit word size, it’s valid to fetch 4 bytes from Mem[0], but not Mem[6] since 6 is not a multiple of 4.
Addresses are indicated by operands that have a bracket “[]” or paren “()”, for Intel vs. AT&T, resp.

What does `mov dl, [al]` do?

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0x0</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>
Addresses are indicated by operands that have a bracket “[]” or paren “()”, for Intel vs. AT&T, resp.

What does `mov edx, [eax]` do?

### Register | Value
--- | ---
eax | 0x3
edx | 0xcc
ebx | 0x5

Which 4 bytes get moved, and which is the LSB in edx?

Address | Value
--- | ---
0x00 | 0
0x0a | 0
0x0b | 0
0x0d | 0
0x0f | 0
0xee | 0
0xff | 0
**Endianess**

- **Endianess**: Order of individually addressable units
- **Little Endian**: Least significant byte first

so address $a$ goes in littlest byte (e.g., AL), $a+1$ in the next (e.g., AH), etc.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

Addr

<table>
<thead>
<tr>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
</tr>
<tr>
<td>0xaa</td>
</tr>
<tr>
<td>0xbb</td>
</tr>
<tr>
<td>0xcc</td>
</tr>
<tr>
<td>0xdd</td>
</tr>
<tr>
<td>0xee</td>
</tr>
<tr>
<td>0xff</td>
</tr>
</tbody>
</table>
`mov edx, [eax]`

**EDX**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td></td>
</tr>
<tr>
<td>edx</td>
<td></td>
</tr>
<tr>
<td>ebx</td>
<td></td>
</tr>
</tbody>
</table>

**EDX = 0xfffeeddcc!**

**Endianess:** Ordering of individually addressable units

**Little Endian:** Least significant byte first

... so ...

address a goes in the least significant byte (the *littlest* bit) a+1 goes into the next byte, and so on.
 mov [eax], ebx

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

**Endianess:** Ordering of individually addressable units

**Little Endian:** Least significant byte first

... so ...

address a goes in the least significant byte (the **littlest** bit) a+1 goes into the next byte, and so on.
There are other ways to address memory than just [register].

These are called *Addressing Modes*.

An *Addressing Mode* specifies how to calculate the effective memory address of an operand by using information from registers and constants contained with the instruction or elsewhere.
Motivation: Addressing Buffers

Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)
Motivation: Addressing Buffers

typedef uint32_t addr_t;
uint32_t w, x, y, z;
uint32_t buf[3] = {1,2,3};
addr_t *ptr = (addr_t) buf;

w = buf[2];
x = *(buf + 2);

What is x? what memory cell does it ref?
Motivation: Addressing Buffers

typedef uint32_t addr_t;
uint32_t w, x, y, z;
uint32_t buf[3] = {1,2,3};
addr_t ptr = (addr_t) buf;

\[
\begin{align*}
w &= buf[2]; \\
x &= *(buf + 2); \\
y &= *( (uint32_t *) (ptr+8));
\end{align*}
\]

Equivalent
*(addr_t) (ptr + 8) = (uint32_t *) buf + 2
Motivation: Addressing Buffers

Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)

Say at \texttt{imm} + \texttt{r}_1

Constant \textit{scaling} factor \texttt{s}, typically 1, 2, 4, or 8

Say in Register \texttt{r}_2

\texttt{imm} + \texttt{r}_1 + \texttt{s}*\texttt{r}_2

AT&T: \texttt{imm (r}_1, \texttt{r}_2, \texttt{s})
Intel: \texttt{r}_1 + \texttt{r}_2*\texttt{s} + \texttt{imm}
# AT&T Addressing Modes for Common Codes

<table>
<thead>
<tr>
<th>Form</th>
<th>Meaning on memory M</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm (r)</td>
<td>M[r + imm]</td>
</tr>
<tr>
<td>imm (r₁, r₂)</td>
<td>M[r₁ + r₂ + imm]</td>
</tr>
<tr>
<td>imm (r₁, r₂, s)</td>
<td>M[r₁ + r₂*s + imm]</td>
</tr>
<tr>
<td>imm</td>
<td>M[imm]</td>
</tr>
</tbody>
</table>
# Address Computation Examples

<table>
<thead>
<tr>
<th>%edx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8 (%edx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80(,%edx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Referencing Memory

Loading a **value** from memory: mov

\[
\texttt{<eax> = *buf;}
\]

\[
\texttt{mov -0x38(%ebp),%eax (A)}
\]
\[
\texttt{mov eax, [ebp-0x38] (I)}
\]

Loading an **address**: lea

\[
\texttt{<eax> = buf;}
\]

\[
\texttt{lea -0x38(%ebp),%eax (A)}
\]
\[
\texttt{lea eax, [ebp-0x38] (I)}
\]
Suppose I want to access address \texttt{0xdeadbeef} directly

**Loads the address**

\texttt{lea eax, 0xdeadbeef (I)}

**Deref the address**

\texttt{mov eax, 0xdeadbeef (I)}

Note missing $. This distinguishes the address from the value.
Understanding Swap

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

**Register Value**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>xp</td>
</tr>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<tr>
<td>%ebx</td>
<td>t0</td>
</tr>
<tr>
<td>%eax</td>
<td>t1</td>
</tr>
</tbody>
</table>

**Commands**

- `movl 8(%ebp), %edx`  # edx = xp
- `movl 12(%ebp), %ecx`  # ecx = yp
- `movl (%edx), %ebx`  # ebx = *xp (t0)
- `movl (%ecx), %eax`  # eax = *yp (t1)
- `movl %eax, (%edx)`  # *xp = t1
- `movl %ebx, (%ecx)`  # *yp = t0

**Stack**

- Offset: 12
  - yp
- Offset: 8
  - xp
- Offset: 4
  - Rtn adr
- Offset: 0
  - Old %ebp
- Offset: -4
  - Old %ebx

%ebp %esp
Understanding Swap

![Diagram of registers and memory addresses]

```
movl 8(%ebp), %edx      # edx = xp
movl 12(%ebp), %ecx    # ecx = yp
movl (%edx), %ebx      # ebx = *xp (t0)
movl (%ecx), %eax      # eax = *yp (t1)
movl %eax, (%edx)      # *xp = t1
movl %ebx, (%ecx)      # *yp = t0
```
Understanding Swap

| %eax   | 0x120 |
| %edx   | 0x124 |
| %ecx   |       |
| %ebx   |       |
| %esi   |       |
| %edi   |       |
| %esp   |       |
| %ebp   | 0x104 |

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>123</td>
</tr>
<tr>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>0x11c</td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

```assembly
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
```
# Understanding Swap

## Variables

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x124</td>
</tr>
<tr>
<td>%edx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

## Offsets

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
</tbody>
</table>

## Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td></td>
</tr>
<tr>
<td>0x120</td>
<td></td>
</tr>
<tr>
<td>0x11c</td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td>0x120</td>
</tr>
<tr>
<td>0x10c</td>
<td>0x124</td>
</tr>
<tr>
<td>0x108</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x104</td>
<td>0x108</td>
</tr>
<tr>
<td>0x100</td>
<td>0x104</td>
</tr>
</tbody>
</table>

## Code

```assembly
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
```
Understanding Swap

\[
\begin{array}{|c|c|}
\hline
%eax & 0x124 \\
%edx & 0x120 \\
%ecx & 123 \\
%ebx & 0x104 \\
%esi & \\
%edi & \\
%esp & 0x100 \\
%ebp & 0x104 \\
\hline
\end{array}
\]

Offset

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>0x114</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{movl} & \ 8(\%ebp), \ %edx \quad \# \ edx = \ xp \\
\text{movl} & \ 12(\%ebp), \ %ecx \quad \# \ ecx = \ yp \\
\text{movl} & \ (%edx), \ %ebx \quad \# \ ebx = *xp \ (t0) \\
\text{movl} & \ (%ecx), \ %eax \quad \# \ eax = *yp \ (t1) \\
\text{movl} & \ %eax, \ (%edx) \quad \# \ *xp = \ t1 \\
\text{movl} & \ %ebx, \ (%ecx) \quad \# \ *yp = \ t0
\end{align*}
\]
Understanding Swap

Offset

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
</tr>
<tr>
<td>0x118</td>
<td>4</td>
</tr>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
</tr>
<tr>
<td>0x118</td>
<td>4</td>
</tr>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

%eax | 456
%edx | 0x124
%ecx | 0x120
%ebx | 123
%esi | 0
%edi | 0
%esp | 0x104
%ebp | 0x100

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
## Understanding Swap

### Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

### Address

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>0x120</td>
</tr>
<tr>
<td>xp</td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
</tr>
</tbody>
</table>

### Instructions

```
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
```
Understanding Swap

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

1. `movl 8(%ebp), %edx # edx = xp`
2. `movl 12(%ebp), %ecx # ecx = yp`
3. `movl (%edx), %ebx # ebx = *xp (t0)`
4. `movl (%ecx), %eax # eax = *yp (t1)`
5. `movl %eax, (%edx) # *xp = t1`
6. `movl %ebx, (%ecx) # *yp = t0`
Agenda

• Compilation Workflow
• x86 Execution Model
  – Basic Execution
  – Memory Operation
  – Control Flow
  – Memory Organization
Assembly is “Spaghetti Code”

**Nice C Abstractions**
- if-then-else
- while
- for loops
- do-while

**Assembly**
- Jump
  - Direct: jmp addr
  - Indirect: jmp reg
- Branch
  - Test EFLAG
  - if(EFLAG SET) goto line
"For" $\rightarrow$ "While" $\rightarrow$ "Do-While"

**For Version**

```plaintext
for (Init; Test; Update)
    Body
```

**While Version**

```plaintext
Init;
while (Test) {
    Body
    Update;
}
```

**Do-While Version**

```plaintext
Init;
if (!Test)
    goto done;
do {
    Body
    Update;
} while (Test)
done:
```

**Goto Version**

```plaintext
Init;
if (!Test)
    goto done;
loop:
    Body
    Update;
    if (Test)
        goto loop;
done:
```
Jump Table

Table Contents

```
.section .rodata
.align 4
.L62:
.long   .L61  # x = 0
.long   .L56  # x = 1
.long   .L57  # x = 2
.long   .L58  # x = 3
.long   .L61  # x = 4
.long   .L60  # x = 5
.long   .L60  # x = 6
```

```
switch(x) {
    case 1:       // .L56
        w = y*z;
        break;
    case 2:       // .L57
        w = y/z;
        /* Fall Through */
    case 3:       // .L58
        w += z;
        break;
    case 5:
    case 6:       // .L60
        w -= z;
        break;
    default:      // .L61
        w = 2;
}
```
Jumps

- jmp 0x45, called a **direct jump**
- jmp *eax, called an **indirect jump**

Branches

- if (EFLAG) jmp x
  Use one of the 32 EFLAG bits to determine if jump taken

**Note:**
No direct way to get or set EIP
Setting EFLAGS

- Instructions may set an eflag, e.g.,
- “cmp” and arithmetic instructions most common
  - Was there a carry (CF Flag set)
  - Was the result zero (ZF Flag set)
  - What was the parity of the result (PF flag)
  - Did overflow occur (OF Flag)
  - Is the result signed (SF Flag)
Aside: Although the x86 processor knows every time integer overflow occurs, C does not make this result visible.

From the Intel x86 manual
See the x86 manuals available on Intel’s website for more information

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>JO</td>
<td>Jump if overflow</td>
<td>OF == 1</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if not overflow</td>
<td>OF == 0</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if sign</td>
<td>SF == 1</td>
</tr>
<tr>
<td>JZ</td>
<td>Jump if zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>JE</td>
<td>Jump if equal</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>JL</td>
<td>Jump if less than</td>
<td>SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JLE</td>
<td>Jump if less than or equal</td>
<td>ZF == 1 or SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if below</td>
<td>CF == 1</td>
</tr>
<tr>
<td>JP</td>
<td>Jump if parity</td>
<td>PF == 1</td>
</tr>
</tbody>
</table>
Agenda

• Compilation Workflow
• x86 Execution Model
  – Basic Execution
  – Memory Operation
  – Control Flow
  – Memory Organization
Memory
Program text
Shared libs
Data
...

The Stack grows down towards lower addresses.

user stack

%esp

shared libraries

brk

run time heap

0xC0000000
(3GB)

- Stack grows down
- Heap grows up

0x00000000
Variables

• On the stack
  – Local variables
  – Lifetime: stack frame

• On the heap
  – Dynamically allocated via new/malloc/etc.
  – Lifetime: until freed
Procedures

• Procedures are not native to assembly
• Compilers *implement* procedures
  – On the stack
  – Following the call/return stack discipline
Procedures/Functions

• We need to address several issues:
  1. How to allocate space for local variables
  2. How to pass parameters
  3. How to pass return values
  4. How to share 8 registers with an infinite number of local variables

• A stack frame provides space for these values
  – Each procedure invocation has its own stack frame
  – Stack discipline is LIFO
    • If procedure A calls B, B’s frame must exit before A’s
orange(...) {
  ...
  red()
  ...
}

red(...) {
  ...
  green()
  ...
  green()
}

green(...) {
  ...
  green()
  ...
}

Function Call Chain

orange → red → green → green → ...

Frame for
- locals
- pushing parameters
- temporary space

Call to red

“**pushes**”

new frame

When green returns it

“**pops**”

its frame

Function **Call Chain**

orange

red

green

green

...
On the stack

```c
int orange(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```

- Need to access arguments
- Need space to store local vars (buf, c, and d)
- Need space to put arguments for callee
- Need a way for callee to return values

Calling convention determines the above features
cdecl – the default for Linux & gcc

int orange(int a, int b) {
    char buf[16];
    int c, d;
    if(a > b) {
        c = a;
    } else {
        c = b;
    }
    d = red(c, buf);
    return d;
}

Don’t worry! We will walk through these one by one.
When orange attains control,
1. return address has already been pushed onto stack by caller
When orange attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8
When *orange* attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8
3. save values of other callee-save registers *if used*
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic
When orange attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8
3. save values of other callee-save registers if used
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic
4. allocate space for locals
   - subtracting from esp
   - “live” variables in registers, which on contention, can be “spilled” to stack space
For *caller orange* to call *callee red*,

```
... 
| b  |
| a  |
| return addr |
| caller’s ebp |
| callee-save |
| locals       |
  (buf, c, d ≥ 24 bytes if stored on stack) |
```

%ebp

%esp
For *caller orange* to call *callee red*,
1. push any caller-save registers if their values are needed after 
   *red* returns
   - eax, edx, ecx
For *caller orange* to call *callee red*,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx

2. push arguments to red from right to left (reversed)
   - from callee’s perspective, argument 1 is nearest in stack
For *caller orange* to call *callee red*,

1. push any caller-save registers if their values are needed after *red* returns
   - eax, edx, ecx
2. push arguments to *red* from right to left (reversed)
   - from callee’s perspective, argument 1 is nearest in stack
3. push return address, i.e., the *next* instruction to execute in *orange* after *red* returns
For *caller orange* to call *callee red*,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx
2. push arguments to red from right to left (reversed)
   - from callee’s perspective, argument 1 is nearest in stack
3. push return address, i.e., the *next* instruction to execute in orange after red returns
4. transfer control to red
   - usually happens together with step 3 using call
When red attains control,

1. return address has already been pushed onto stack by orange

```
<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>return addr</td>
</tr>
<tr>
<td>caller’s ebp</td>
</tr>
<tr>
<td>callee-save</td>
</tr>
<tr>
<td>locals (buf, c, d ≥ 24 bytes if stored on stack)</td>
</tr>
<tr>
<td>caller-save</td>
</tr>
<tr>
<td>buf</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>return addr</td>
</tr>
</tbody>
</table>
```

%ebp  %esp
When red attains control,
1. return address has already been pushed onto stack by orange
2. own the frame pointer
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. ... (red is doing its stuff) ...

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

%ebp ———— %esp 76
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. ... (red is doing its stuff) ...
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
When red attains control,
1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange’s frame pointer
   - pop %ebp
When red attains control,
1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. ... (red is doing its stuff) ...
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange’s frame pointer
   - pop %ebp
8. return control to orange
   - ret
   - pops return address from stack and jumps there
When orange regains control,
When orange regains control,
1. clean up arguments to red  
   - adding to esp
2. restore any caller-save registers  
   - pops
3. ...
Terminology

- **Function Prologue** – instructions to set up stack space and save callee saved registers
  - Typical sequence:
    - push ebp
    - ebp = esp
    - esp = esp - <frame space>

- **Function Epilogue** - instructions to clean up stack space and restore callee saved registers
  - Typical Sequence:
    - leave  // esp = ebp, pop ebp
    - ret    // pop and jump to ret addr
# cdecl – One Convention

<table>
<thead>
<tr>
<th>Action</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller saves: eax, edx, ecx</td>
<td>push (old), or mov if esp already adjusted</td>
</tr>
<tr>
<td>arguments pushed right-to-left</td>
<td></td>
</tr>
<tr>
<td>linkage data starts new frame</td>
<td>call pushes return addr</td>
</tr>
<tr>
<td>callee saves: ebx, esi, edi, ebp, esp</td>
<td>ebp often used to deref args and local vars</td>
</tr>
<tr>
<td>return value</td>
<td>pass back using eax</td>
</tr>
<tr>
<td>argument cleanup</td>
<td>caller’s responsibility</td>
</tr>
</tbody>
</table>
**Revisiting swap**

```c
int course1 = 15213;
int course2 = 18243;

void call_swap() {
    swap(&course1, &course2);
}

void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

**Calling swap from call_swap**

```c
call_swap:
    ....
    subl $8, %esp
    movl $course2, 4(%esp)
    movl $course1, (%esp)
    call swap
    ....
```

**Resulting Stack**

```
%esp
&course2
&course1
Rtn adr
```

---

`sbl $8, %esp` subtracts 8 from the stack pointer `%esp`.

`movl $course2, 4(%esp)` moves the value of `course2` to the memory location 4 bytes below the current `%esp`.

`movl $course1, (%esp)` moves the value of `course1` to the memory location pointed to by `%esp`.

`call swap` calls the `swap` function.

---

The diagram shows the stack layout after the variables are swapped:

- `%esp` (subtracted by 8)
- `&course2` (4 bytes below `%esp`)
- `&course1` (pointed to by `%esp`)
- `Rtn adr` (return address at `%esp`)

---

The swap function works as follows:

1. `t0 = *xp;` (copy value of `xp` to `t0`)
2. `t1 = *yp;` (copy value of `yp` to `t1`)
3. `*xp = t1;` (assign `t1` to `xp`)
4. `*yp = t0;` (assign `t0` to `yp`)

This results in the values of `xp` and `yp` being swapped.
Revisiting swap

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

**swap:**

```assembly
pushl %ebp
movl %esp, %ebp
pushl %ebx

movl 8(%ebp), %edx
movl 12(%ebp), %ecx
movl (%edx), %ebx
movl (%ecx), %eax
movl %eax, (%edx)
movl %ebx, (%ecx)

popl %ebx
popl %ebp
ret
```

- **Set Up**
- **Body**
- **Finish**
swap Setup #1

Entering Stack

Resulting Stack

\[
\text{swap:} \\
\begin{align*}
\text{pushl } &\%ebp \\
\text{movl } &\%esp,\%ebp \\
\text{pushl } &\%ebx
\end{align*}
\]
swap Setup #2

**Entering Stack**

- %ebp
- &course2
- &course1
- Rtn adr

**Resulting Stack**

- %ebp
- yp
- xp
- Rtn adr
- Old %ebp

**swap:**

```assembly
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
swap Setup #3

### Entering Stack

- `%ebp`
- `&course2`
- `&course1`
- `Rtn adr`

### Resulting Stack

- `yp`
- `xp`
- `Rtn adr`
- `Old %ebp`
- `Old %ebx`
- `%esp`
- `%ebp`

---

```
swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
swap Body

Entering Stack

•
•
•

Resulting Stack

•
•

Offset relative to %ebp

&course2

%ebp →

12

&course1

%esp

4

Rtn adr

Rtn adr

Old %ebp

Old %ebx

movl 8(%ebp),%edx  # get xp
movl 12(%ebp),%ecx  # get yp
. . .
swap Finish

Stack Before Finish

Resulting Stack

Observation
- Saved and restored register %ebx
- Not so for %eax, %ecx, %edx
Disassembled swap

08048384 <swap>:

8048384:  55                  push   %ebp
8048385:  89 e5               mov    %esp,%ebp
8048387:  53                  push   %ebx
8048388:  8b 55 08            mov    0x8(%ebp),%edx
804838b:  8b 4d 0c            mov    0xc(%ebp),%ecx
804838e:  8b 1a               mov    (%edx),%ebx
8048390:  8b 01               mov    (%ecx),%eax
8048392:  89 02               mov    %eax,(%edx)
8048394:  89 19               mov    %ebx,(%ecx)
8048396:  5b                  pop    %ebx
8048397:  5d                  pop    %ebp
8048398:  c3                  ret

Calling Code

80483b4:  movl   $0x8049658,0x4(%esp)  # Copy &course2
80483bc:  movl   $0x8049654,(%esp)     # Copy &course1
80483c3:  call   8048384 <swap>       # Call swap
80483c8:  leave                 # Prepare to return
80483c9:  ret                   # Return
IA32/Linux+Windows Register Usage

- **%eax, %edx, %ecx**
  - Caller saves prior to call if values are used later

- **%eax**
  - also used to return integer value

- **%ebx, %esi, %edi**
  - Callee saves if wants to use them

- **%esp, %ebp**
  - special form of callee save
  - Restored to original values upon exit from procedure
Control Flow: Function Calls

- What must assembly/machine language do?

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
</table>
| 1. Save function arguments  
2. Branch to function body | 3. Execute body  
- May allocate memory  
- May call functions  
4. Save function result  
5. Branch to where called |

1. Use registers `%edi, %esi, etc., then stack
2. Use call (jump to procedure, save return location on stack)
3. Use `sub %esp` to create new stack frame
4. Use `%eax`
5. Use `ret`
Agenda

• Compilation Workflow
• x86 Execution Model
  – Basic Execution
  – Memory Operation
  – Control Flow
  – Memory Organization
For more information

• Overall machine model: *Computer Systems, a Programmer’s Perspective* by Bryant and O’Hallaron

• Calling Conventions:
Questions